Clocks as Types in Synchronous Dataflow Languages

Marc Pouzet
LRI & INRIA
Univ. Paris-Sud 11
Orsay

IFIP WG 2.8 – 9/06/2009

(joint work with Albert Cohen, Louis Mandel, Florence Plateau)

Synchronous Dataflow Languages

Model/program critical embedded software.

The idea of Lustre :

- directly write stream equations as **executable specifications**
- provide a **compiler** and associated analyzing tools to generate embedded code

E.g, the linear filter :

$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$

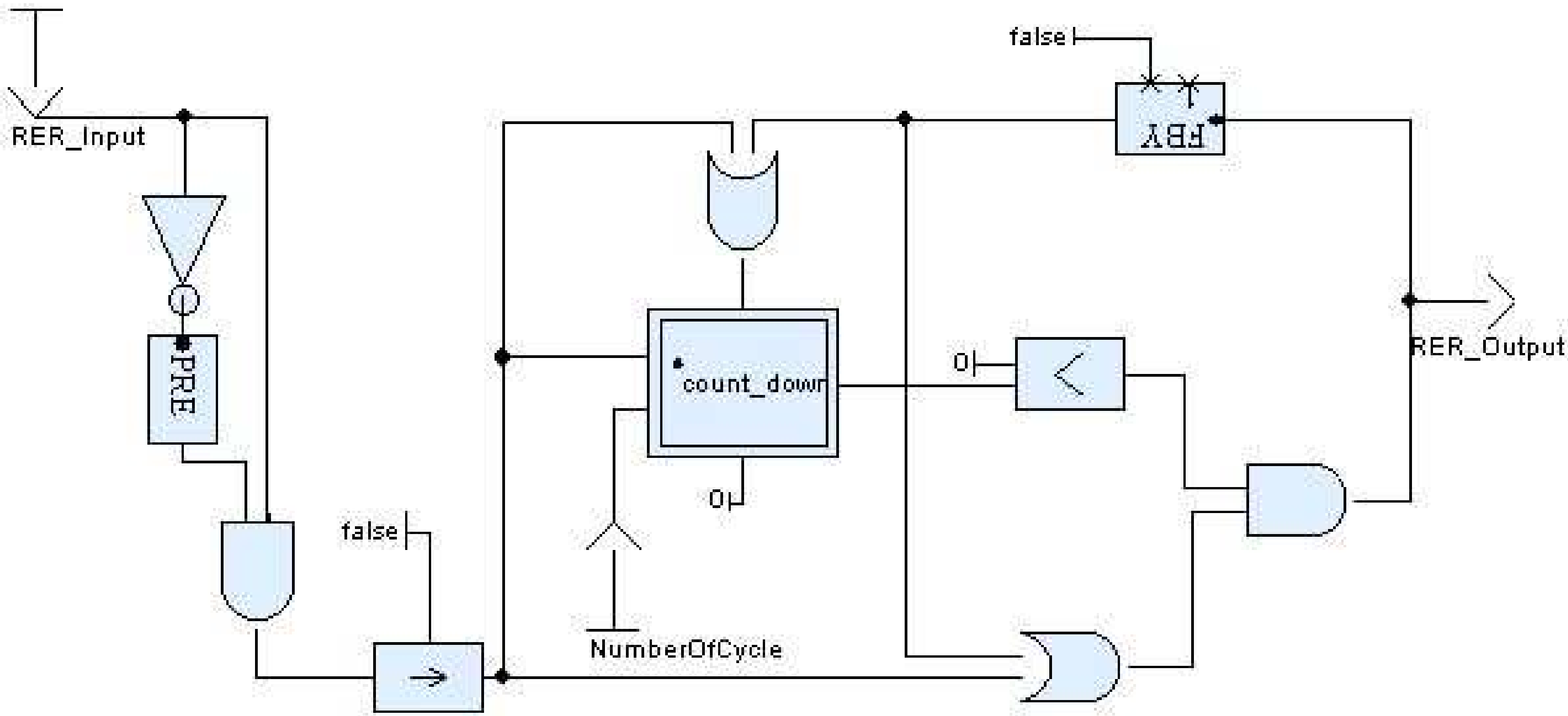
is programmed by writing, e.g :

```
Y = (0 -> a * pre(Y)) + Z;
Z = b * X
```

we write **invariants**

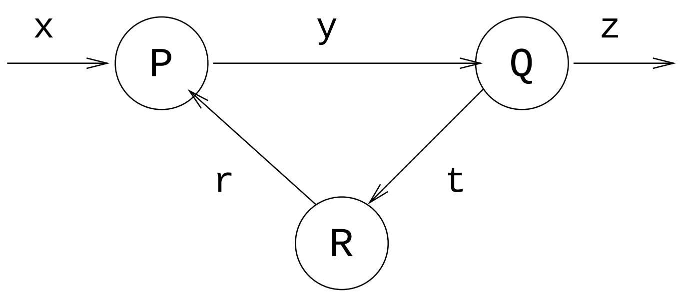
- other primitives to deal with slow and fast processes (sub/over-sampling) ; not necessarily periodic

An example of a SCADE sheet



Dataflow Semantics

Kahn Principle : The semantics of process networks communicating through unbounded FIFOs (e.g., Unix pipe, sockets) ?



– message communication into FIFOs (`send`/`wait`)

– reliable channels, bounded communication delay

– blocking wait on a channel. The following program is **forbidden**

```
if (A is present) or (B is present) then ...
```

– a process = a continuous function $(V^{\infty})^n \rightarrow (V'^{\infty})^m$.

Lustre :

– Lustre has a **Kahn semantics** (no test of absence)

– A dedicated **type system** (clock calculus) to guaranty the existence of an execution with no buffer (no synchronization)

Pros and Cons of KPN

(+) : Simple semantics : a process defines a function (determinism) ; composition is function composition

(+) : Modularity : a network is a continuous function

(+) : Asynchronous distributed execution : easy ; no centralized scheduler

(+/-) : Time invariance : no explicit timing ; but impossible to state that two events happen at the same time.

x	$=$	x_0	x_1		x_2	x_3	x_4	x_5			$\ldots$
$f(x)$	$=$	y_0	y_1		y_2	y_3	y_4	y_5			$\ldots$
$f(x)$	$=$	y_0		y_1	y_2		y_3		y_4	y_5	$\ldots$

This appeared to be a useful model for video apps (TV boxes) : Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) with various "synchronous" restriction *à la SDF* (Edward Lee)

A small dataflow kernel

A small kernel with minimal primitives

$$
\begin{array}{rcl}
e & ::= & e\ \texttt{fby}\ e \mid op(e, ..., e) \mid x \mid i \\
 & & \mid \texttt{merge}\ e\ e\ e \mid e\ \texttt{when}\ e \\
 & & \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x.e \\
op & ::= & + \mid - \mid \texttt{not} \mid ...
\end{array}
$$

- function ($\lambda x.e$), application ($e\ e$), fix-point ($\texttt{rec}\ x.e$)
- constants i and variables (x)
- dataflow primitives : x `fby` y is the unitary delay ; $op(e_1, ..., e_n)$ the point-wise application ; sub-sampling/oversampling (`when`/`merge`).

Dataflow Primitives

x	x_0	x_1	x_2	x_3	x_4	x_5
y	y_0	y_1	y_2	y_3	y_4	y_5
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
x `fby` y	x_0	y_0	y_1	y_2	y_3	y_4
h	1	0	1	0	1	0
$x' = x$ `when` h	x_0		x_2		x_4	
z		z_0		z_1		z_2
`merge` $h\ x'\ z$	x_0	z_0	x_2	z_1	x_4	z_2

Sampling :

- if h is a boolean sequence, x `when` h produces a sub-sequence of x
- `merge` $h\ x\ z$ combines two sub-sequences

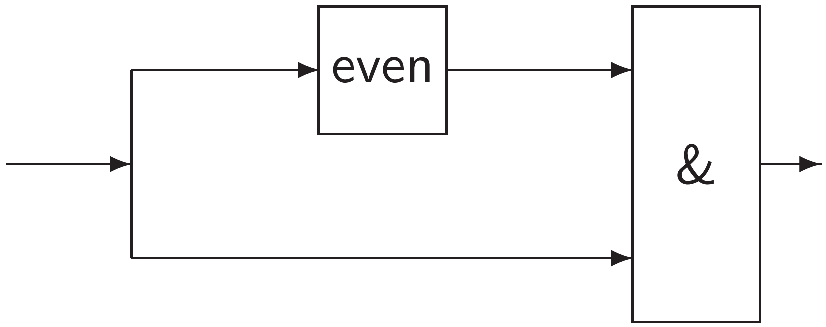
Kahn Semantics

Every operator is interpreted as a stream function $(V^{\infty} = V^{*} + V^{\omega})$. E.g., if $x \mapsto s_1$ and $y \mapsto s_2$ then the value of $x + y$ is $+^{\#}(s_1, s_2)$

$$
\begin{array}{lcl}
i^{\#} & = & i.i^{\#} \\
\\
+^{\#}(x.s_1, y.s_2) & = & (x+y).+^{\#}(s_1, s_2) \\
\\
(x.s_1)\,\texttt{fby}^{\#}\,s_2 & = & x.s_2 \\
\\
x.s\,\texttt{when}^{\#}\,1.c & = & x.(s\,\texttt{when}^{\#}\,c) \\
x.s\,\texttt{when}^{\#}\,0.c & = & s\,\texttt{when}^{\#}\,c \\
\\
\texttt{merge}^{\#}\,1.c\,x.s_1\,s_2 & = & x.\texttt{merge}^{\#}\,c\,s_1\,s_2 \\
\texttt{merge}^{\#}\,0.c\,s_1\,y.s_2 & = & y.\texttt{merge}^{\#}\,c\,s_1\,s_2
\end{array}
$$

Synchrony

Some programs generate monsters.



If $x = (x_i)_{i \in \mathbb{N}}$ then $\mathtt{even}(x) = (x_{2i})_{i \in \mathbb{N}}$ and $x \& \mathtt{even}(x) = (x_i \& x_{2i})_{i \in \mathbb{N}}$.

Unbounded FIFOs !

- must be rejected statically
- every operator is finite memory through the composition is not : all the complexity (synchronization) is hidden in communication channels
- the Kahn semantics does not model time, i.e., impossible to state that two event arrive **at the same time**

Synchronous (Clocked) streams

Complete streams with an explicit representation of absence (abs).

$$x : (V^{abs})^{\infty}$$

Clock : the clock of x is a boolean sequence

$$\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}$$

$$\mathcal{CLOCK} = \mathbb{B}^{\infty}$$

$$\begin{array}{lcl} \mathtt{clock}\ \epsilon & = & \epsilon \\ \mathtt{clock}\ (abs.x) & = & \mathtt{0.clock}\ x \\ \mathtt{clock}\ (v.x) & = & \mathtt{1.clock}\ x \end{array}$$

Synchronous streams :

$$ClStream(V, cl) = \{s/s \in (V^{abs})^{\infty} \wedge \mathtt{clock}\ s \leq_{prefix} cl\}$$

An other possible encoding : $x : (V \times \mathbb{N})^{\infty}$

Dataflow Primitives

Constant :

$$
\begin{aligned}
i^{\#}(\epsilon) &= \epsilon \\
i^{\#}(1.cl) &= i.i^{\#}(cl) \\
i^{\#}(0.cl) &= abs.i^{\#}(cl)
\end{aligned}
$$

Point-wise application :

Synchronous arguments must be constant, i.e., having the same clock

$$
\begin{aligned}
+^{\#}(s_1, s_2) &= \epsilon \text{ if } s_i = \epsilon \\
+^{\#}(abs.s_1, abs.s_2) &= abs.+^{\#}(s_1, s_2) \\
+^{\#}(v_1.s_1, v_2.s_2) &= (v_1 + v_2).+^{\#}(s_1, s_2)
\end{aligned}
$$

Partial definitions

What happens when one element is present and the other is absent ?

Constraint their domain :

$$(+) : \forall cl : \mathcal{CLOCK}.ClStream(\texttt{int}, cl) \times ClStream(\texttt{int}, cl) \rightarrow ClStream(\texttt{int}, cl)$$

i.e., $(+)$ expect its two input stream to be on the same clock cl and produce an output on the same clock

These extra conditions are **types** which must be statically verified

Remark (notation) : Regular types and clock types can be written separately :

- $(+) : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ $\leftarrow$ **its type**
- $(+) :: \forall cl.cl \times cl \rightarrow cl$ $\leftarrow$ **its clock type**

In the following, we only consider the clock type.

Sampling

$$
\begin{aligned}
s_1 \,\texttt{when}^{\#}\, s_2 &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(abs.s) \,\texttt{when}^{\#}\, (abs.c) &= abs.s \,\texttt{when}^{\#}\, c \\
(v.s) \,\texttt{when}^{\#}\, (\mathsf{1}.c) &= v.s \,\texttt{when}^{\#}\, c \\
(v.s) \,\texttt{when}^{\#}\, (\mathsf{0}.c) &= abs.x \,\texttt{when}^{\#}\, c
\end{aligned}
$$

$$
\begin{aligned}
\texttt{merge}\, c\, s_1\, s_2 &= \epsilon \text{ if one of the } s_i = \epsilon \\
\texttt{merge}\, (abs.c)\, (abs.s_1)\, (abs.s_2) &= abs.\texttt{merge}\, c\, s_1\, s_2 \\
\texttt{merge}\, (\mathsf{1}.c)\, (v.s_1)\, (abs.s_2) &= v.\texttt{merge}\, c\, s_1\, s_2 \\
\texttt{merge}\, (\mathsf{0}.c)\, (abs.s_1)\, (v.s_2) &= v.\texttt{merge}\, c\, s_1\, s_2
\end{aligned}
$$

Examples

$base = (1)$	1	1	1	1	1	1	1	1	1	1	1	1	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	...
$h = (10)$	1	0	1	0	1	0	1	0	1	0	1	0	...
$y = x$ `when` h	x_0		x_2		x_4		x_6		x_8		x_{10}	x_{11}	...
$h' = (100)$	1		0		0		1		0		0	1	...
$z = y$ `when` h'	x_0						x_6					x_{11}	...
k			k_0		k_1				k_2		k_3		...
`merge` $h'\ z\ k$	x_0		k_0		k_1		x_6		k_2		k_3		...

```
let clock five =
  let rec f = true fby false fby false fby false fby f in f
let node stutter x = o where
  rec o = merge five x ((0 fby o) whenot five) in o
```

`stutter`$(nat) = 0.0.0.0.1.1.1.1.2.2.2.2.3.3...$

Sampling and clocks

- $x\,\texttt{when}^{\#}\,y$ is defined when x and y have the same clock cl
- the clock of $x\,\texttt{when}^{\#}\,c$ is written $cl\,\mathsf{on}\,c$: "c moves at the pace of cl"

$$
\begin{array}{lcl}
s\,\mathsf{on}\,c & = & \epsilon \text{ if } s = \epsilon \text{ or } c = \epsilon \\
(1.cl)\,\mathsf{on}\,(1.c) & = & 1.cl\,\mathsf{on}\,c \\
(1.cl)\,\mathsf{on}\,(0.c) & = & 0.cl\,\mathsf{on}\,c \\
(0.cl)\,\mathsf{on}\,(abs.c) & = & 0.cl\,\mathsf{on}\,c
\end{array}
$$

We get :

$$\texttt{when} : \forall cl.\forall x : cl.\forall c : cl.cl\,\mathsf{on}\,c$$

$$\texttt{merge} : \forall cl.\forall c : cl.\forall x : cl\,\mathsf{on}\,c.\forall y : cl\,\mathsf{on}\,\mathit{not}\;c.cl$$

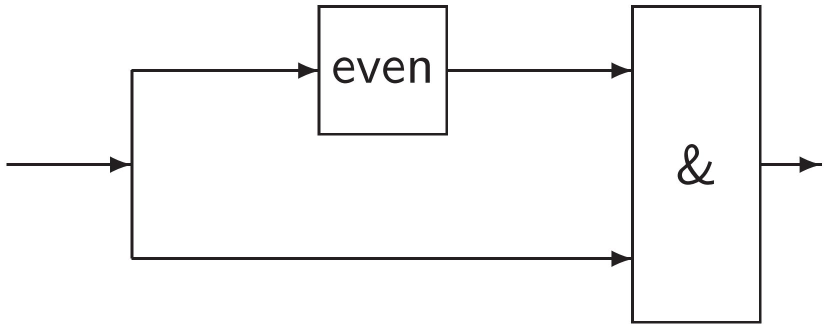
Written instead :

$$\texttt{when} : \forall cl.cl \rightarrow (c : cl) \rightarrow cl\,\mathsf{on}\,c$$

$$\texttt{merge} : \forall cl.(c : cl) \rightarrow cl\,\mathsf{on}\,c \rightarrow cl\,\mathsf{on}\,\mathit{not}\;c \rightarrow cl$$

Checking Synchrony

The previous program is now rejected.



This is a now a **typing error**

```
let even x = x when half
let non_synchronous x = x & (even x)
                            ^^^^^^^^

This expression has clock 'a on half,
but is used with clock 'a
```

Final remarks :

– We only considered **clock equality**, i.e., "two streams are either synchronous or not"
– Clocks are used extensively to generate **efficient sequential code**

From Synchrony to Relaxed Synchrony

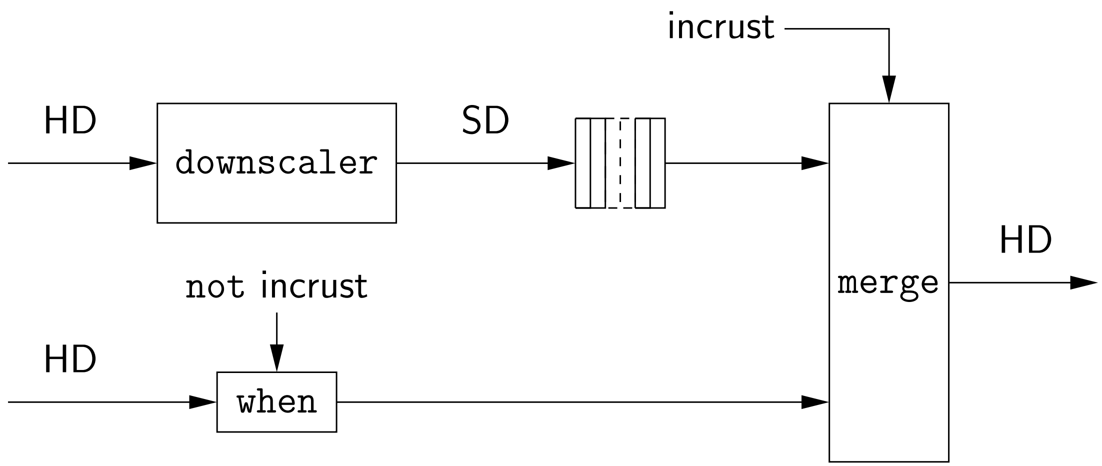
– can we compose non strictly synchronous streams provided their clocks are closed from each other ?

– communication between systems which are “almost” synchronous

– model jittering, bounded delays

– Give more freedom to the compiler, generate more efficient code, translate into regular synchronous code if necessary

A typical example : Picture in Picture



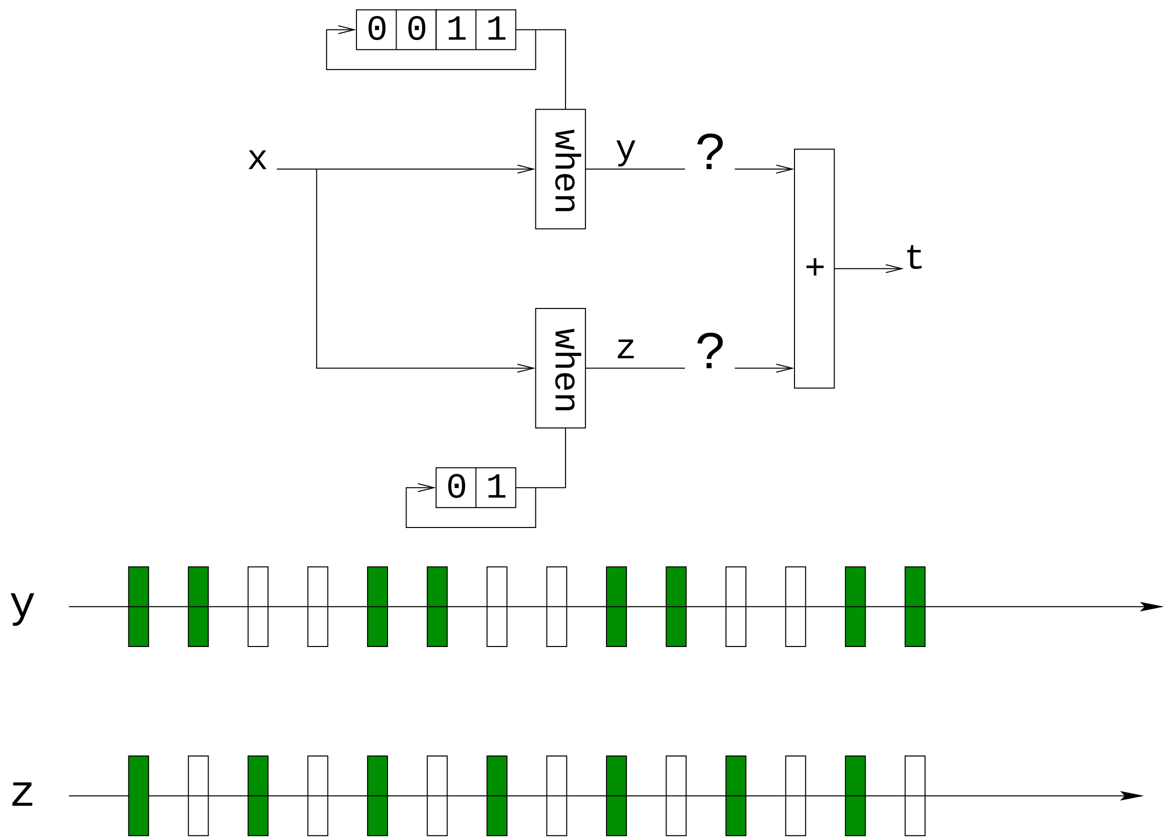
Incrustation of a Standard Definition (SD) image in a High Definition (HD) one

- `downscaler` : reduction of an HD image (1920×1080 pixels) to an SD image (720×480 pixels)
- `when` : removal of a part of an HD image
- `merge` : incrustation of an SD image in an HD image

Question :

- buffer size needed between the `downscaler` and the `merge` nodes ?
- delay introduced by the picture in picture in the video processing chain ?

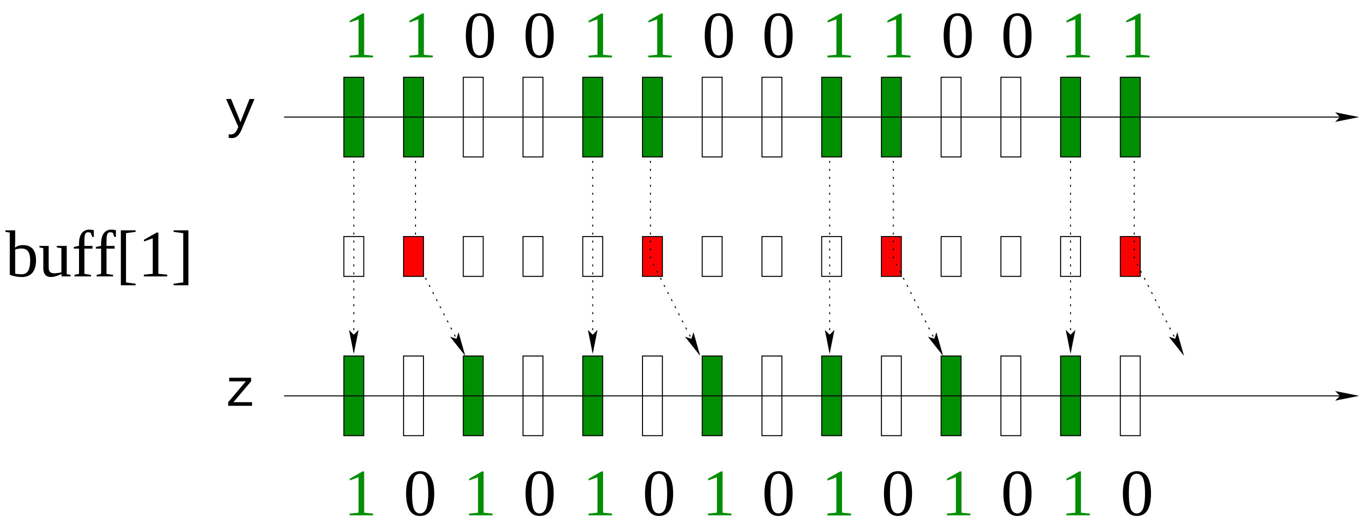
Too restrictive for video applications



- streams should be synchronous
- adding buffer (by hand) difficult and error-prone
- compute it automatically and generate synchronous code

relax the associated clocking rules

N-Synchronous Kahn Networks



– based on the use of *infinite ultimately periodic sequences*

– a precedence relation $cl_1 <: cl_2$

Ultimately periodic sequences

$\mathbb{Q}_2$ for the set of infinite periodic binary words.

$$\begin{array}{rcl} (01) & = & 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01 \ldots \\ 0(1101) & = & 0\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101 \ldots \end{array}$$

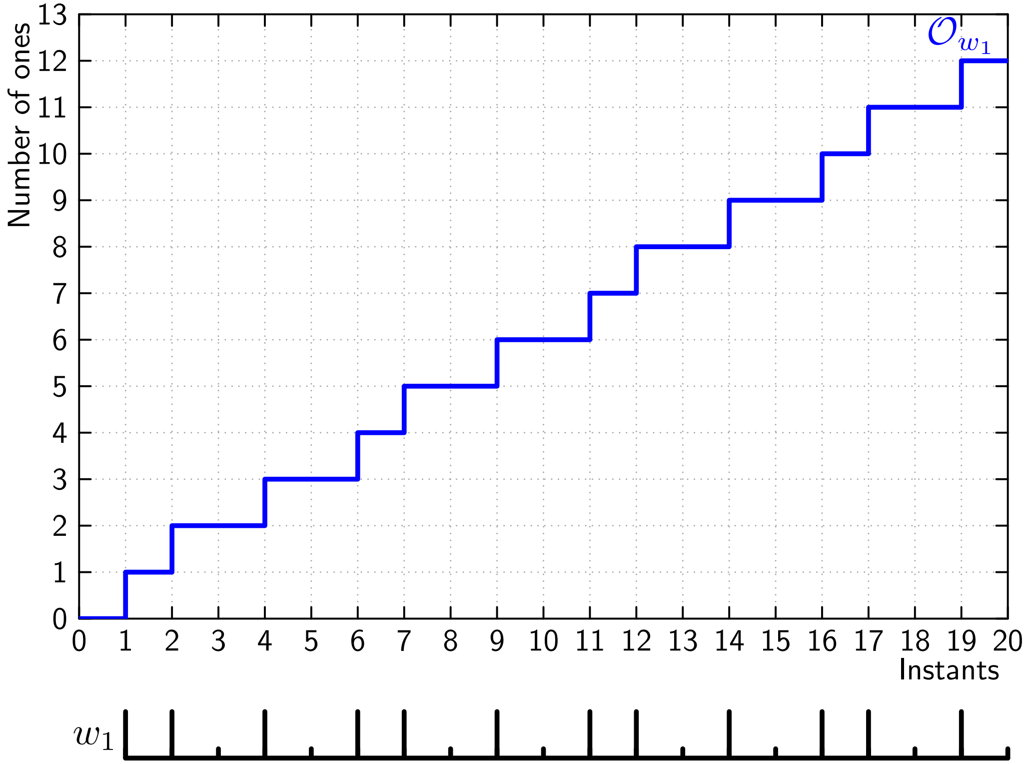
– 1 for presence

– 0 for absence

Definition :

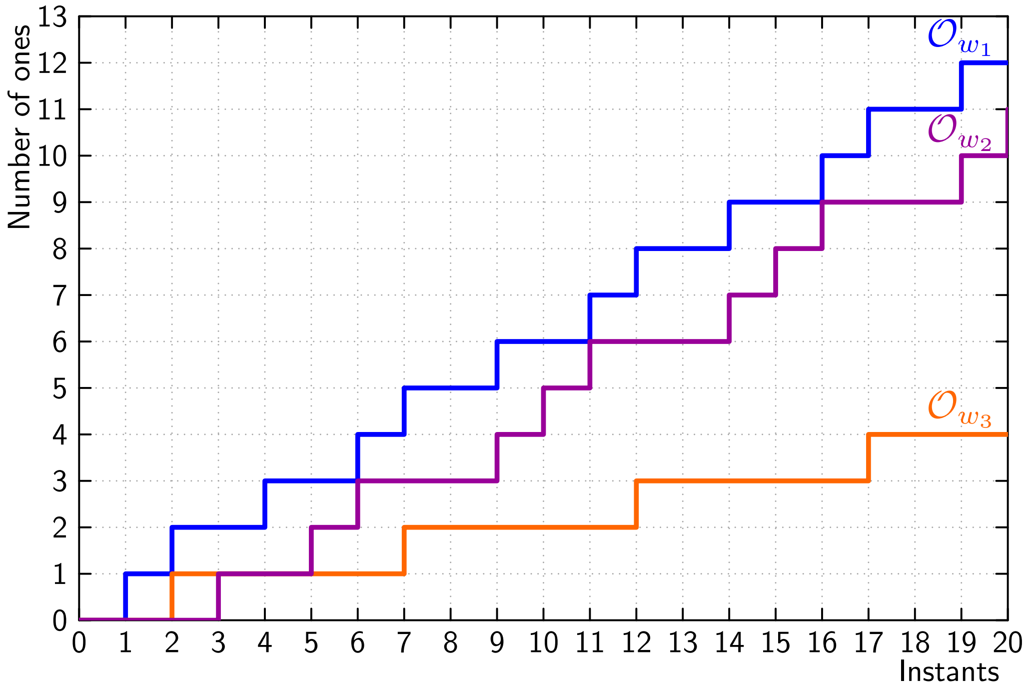
$$w ::= u(v) \quad \text{where } u \in (0+1)^* \text{ and } v \in (0+1)^+$$

Clocks and infinite binary words



$$\mathcal{O}_w(i) = \text{cumulative function of 1 from } w$$

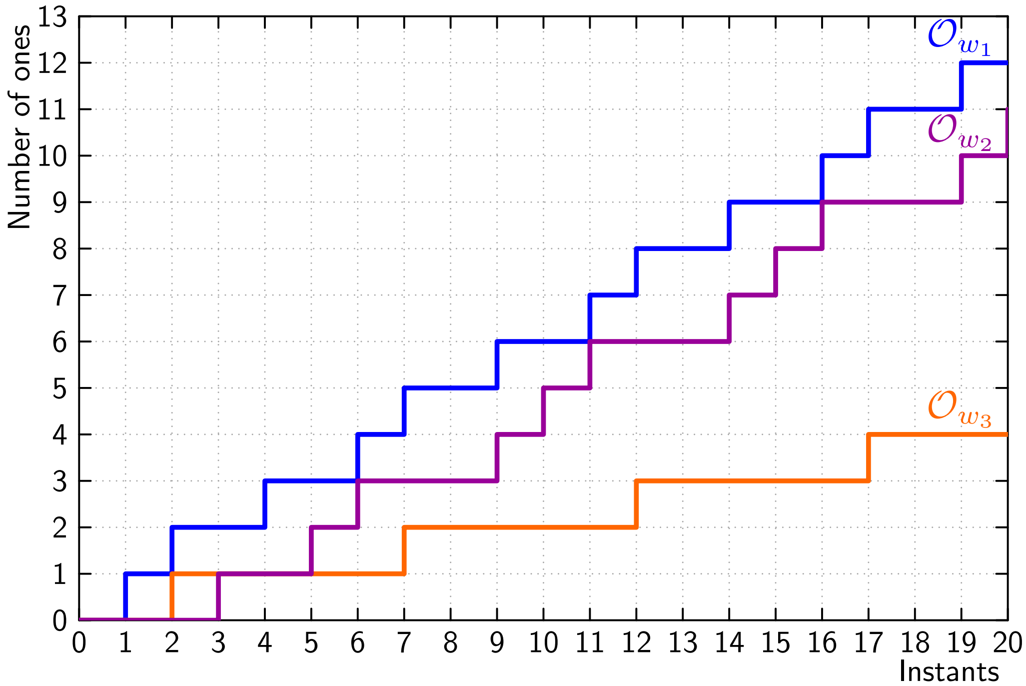
Clocks and infinite binary words



buffer $size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$

sub-typing $w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$

Clocks and infinite binary words



buffer	$size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$
sub-typing	$w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$
synchronizability	$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z}, \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$
precedence	$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$

Multi-clock

$$c \quad ::= \quad w \mid c \texttt{ on } w \qquad w \in (0+1)^\omega$$

c on w is a **sub-clock** of c, by moving in w at the pace of c. E.g., $1(10) \texttt{ on } (01) = (0100)$.

base	1	1	1	1	1	1	1	1	1	1	...	(1)
p_1	1	1	0	1	0	1	0	1	0	1	...	1(10)
base on p_1	1	1	0	1	0	1	0	1	0	1	...	1(10)
p_2	0	1		0		1		0		1	...	(01)
(base on p_1) on p_2	0	1	0	0	0	1	0	0	0	1	...	(0100)

For ultimately periodic clocks, precedence, synchronizability and equality are decidable (but expensive)

Come-back to the language

Pure synchrony :

- close to an ML type system (e.g., SCADE 6)
- structural equality of clocks

$$\frac{H \vdash e_1 : ck \qquad H \vdash e_2 : ck}{H \vdash op(e_1, e_2) : ck}$$

Relaxed Synchrony :

- we add a **sub-typing** rule :

$$(\text{SUB}) \quad \frac{H \vdash e : ck \ \texttt{on} \ w \qquad w <: w'}{H \vdash e : ck \ \texttt{on} \ w'}$$

- defines synchronization points when a buffer is inserted

What about non periodic systems ?

- The same idea : synchrony + properties between clocks. Insuring the absence of deadlocks and bounded buffering.

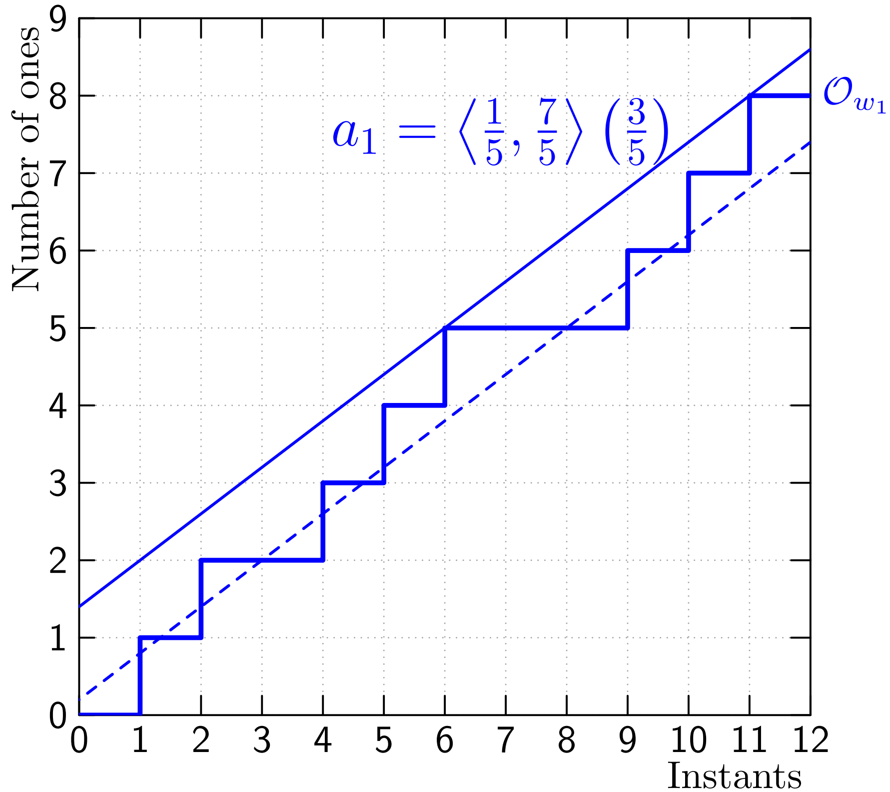
- The **exact** computation with periodic clocks does not work in practice (and is useless). E.g., (10100100) on $0^{3600}(1)$ on $(101001001) =$ $0^{9600}(10^4 10^7 10^7 10^2)$

- Motivations :

  1. To treat long periodic patterns. To avoid an exact computation.
  2. To deal with almost periodic clocks. E.g., α on w where
     $w = 00.(\ (10) + (01)\ )^*$
     (e.g. $w = 00\ 01\ 10\ 01\ 01\ 10\ 01\ 10 \ldots$)

Idea : manipulate sets of clocks ; turn questions into arithmetic ones

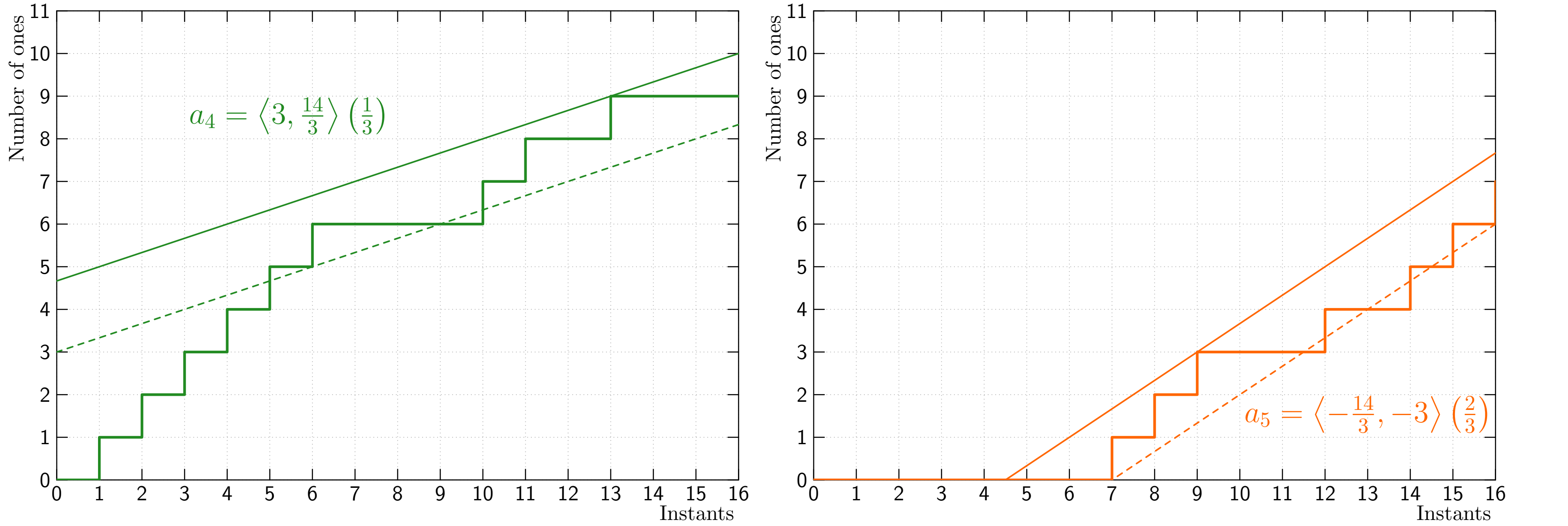
Abstraction of Infinite Binary Words



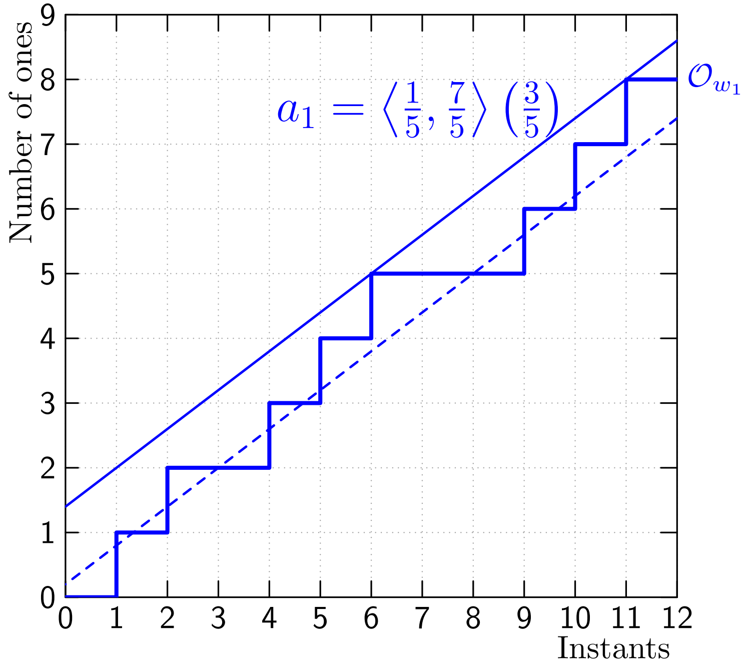
A word w can be abstracted by two lines : $abs(w) = \left\langle b^0, b^1 \right\rangle (r)$

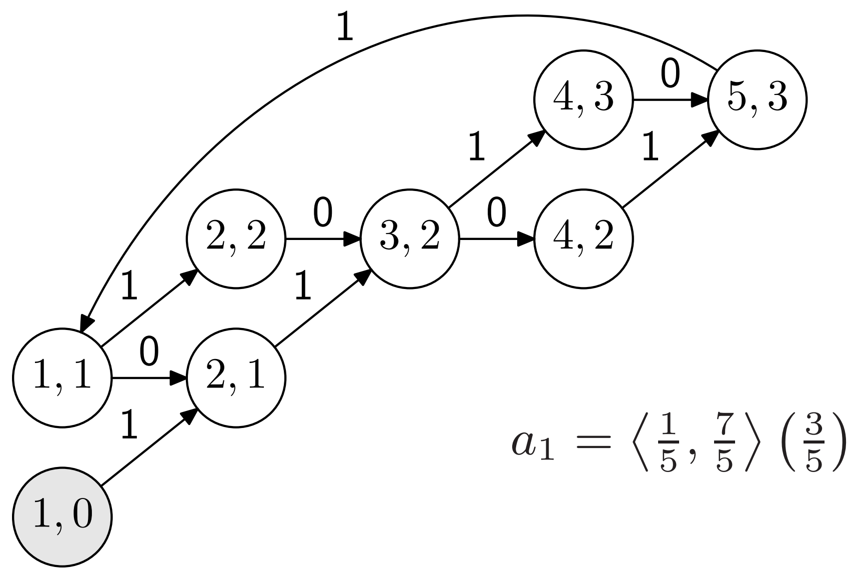
$$concr\left(\left\langle b^0, b^1 \right\rangle (r)\right) \overset{def}{\Leftrightarrow} \left\{ w,\ \forall i \geq 1,\ \wedge \begin{array}{lll} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

Abstraction of Infinite Binary Words



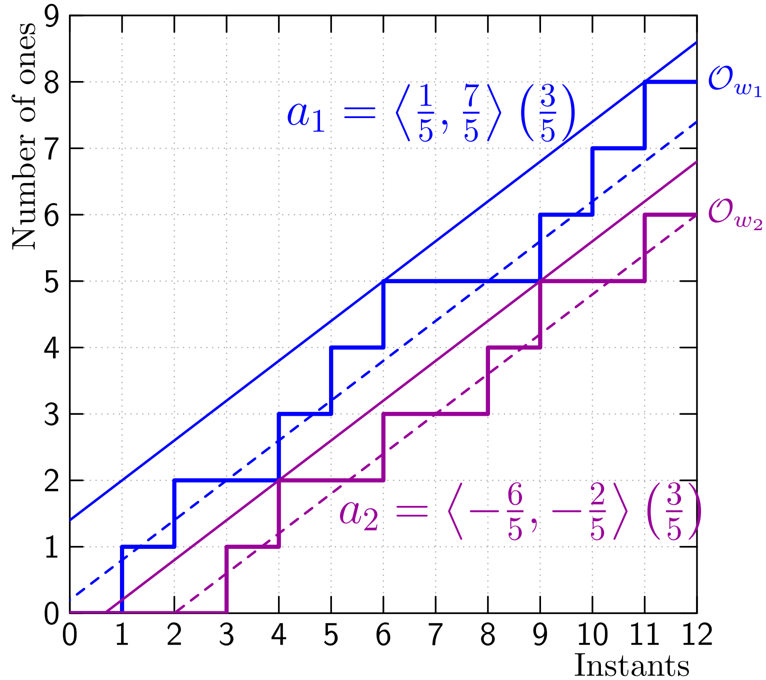
Abstract Clocks as Automata





- set of states $\{(i, j) \in \mathbb{N}^2\}$: coordinates in the 2D-chronogram
- finite number of state equivalence classes
- transition function δ : $\begin{cases} \delta(1,(i,j)) = nf(i+1, j+1) & \text{if } j+1 \leq r \times i + b^1 \\ \delta(0,(i,j)) = nf(i+1, j+0) & \text{if } j+0 \geq r \times i + b^0 \end{cases}$
- allows to check/generate clocks

Abstract Relations



Synchronizability : $r_1 = r_2 \Leftrightarrow \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) \bowtie^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2)$

Precedence : ${b^1}_2 - {b^0}_1 < 1 \Rightarrow \left\langle {b^0}_1, {b^1}_1 \right\rangle (r) \preceq^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r)$

Subtyping : $a_1 <:^{\sim} a_2 \Leftrightarrow a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2$

▷ proposition : $abs(w_1) <:^{\sim} abs(w_2) \Rightarrow w_1 <: w_2$

▷ buffer : $size(a_1, a_2) = \left\lfloor {b^1}_1 - {b^0}_2 \right\rfloor$

Abstract Operators

Composed clocks : $c ::= w \mid \mathbf{not}\ w \mid c\ \mathbf{on}\ c$

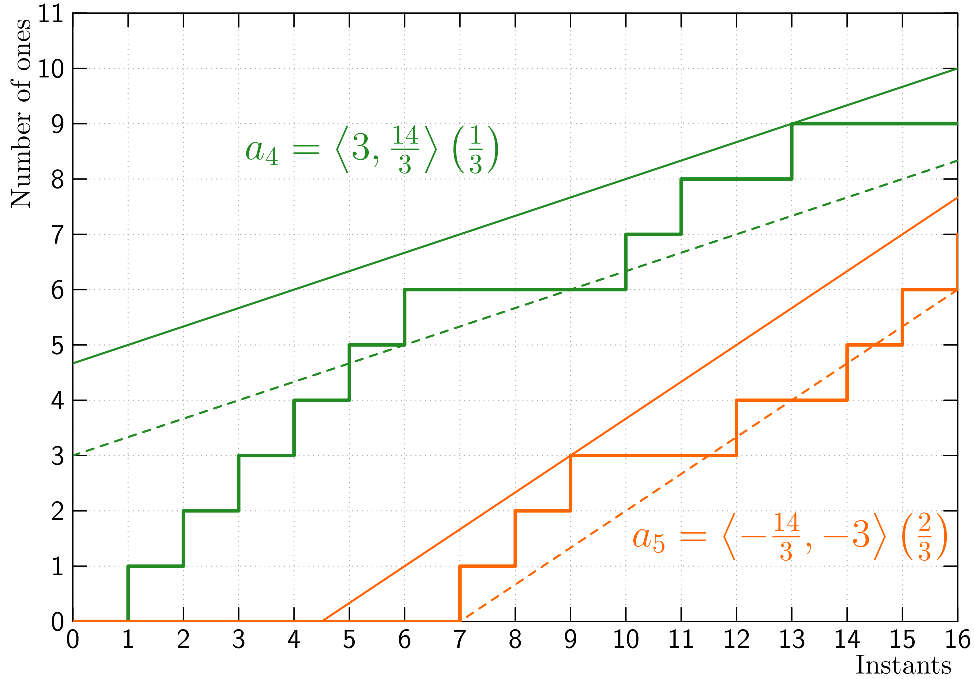
Abstraction of a composed clock :

$$\begin{array}{rcl} abs(\mathbf{not}\ w) & = & \mathbf{not}^{\sim}\ abs(w) \\ abs(c_1\ \mathbf{on}\ c_2) & = & abs(c_1)\ \mathbf{on}^{\sim}\ abs(c_2) \end{array}$$

Operators correctness property :

$$\begin{array}{rcl} \mathbf{not}\ w & \in & concr(\mathbf{not}^{\sim}\ abs(w)) \\ c_1\ \mathbf{on}\ c_2 & \in & concr(abs(c_1)\ \mathbf{on}^{\sim}\ abs(c_2)) \end{array}$$

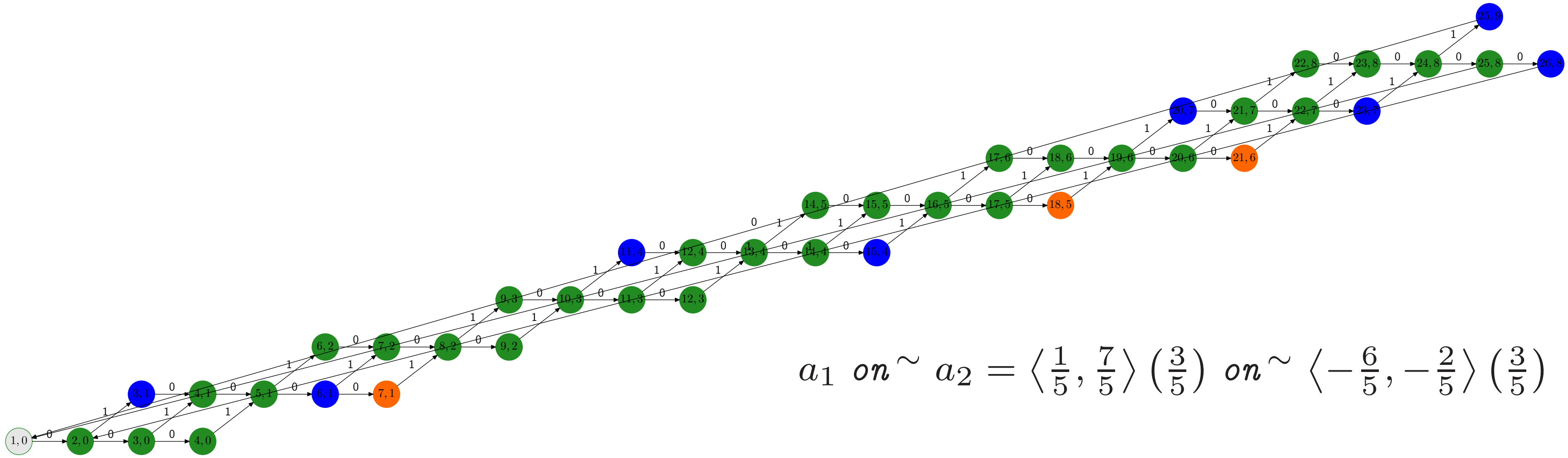
Abstract Operators



$\mathit{not}^{\sim}$ operator definition :

- $\mathit{not}^{\sim} \left\langle b^0, b^1 \right\rangle (r) = \left\langle -b^1, -b^0 \right\rangle (\ 1 - r)$

Abstract Operators



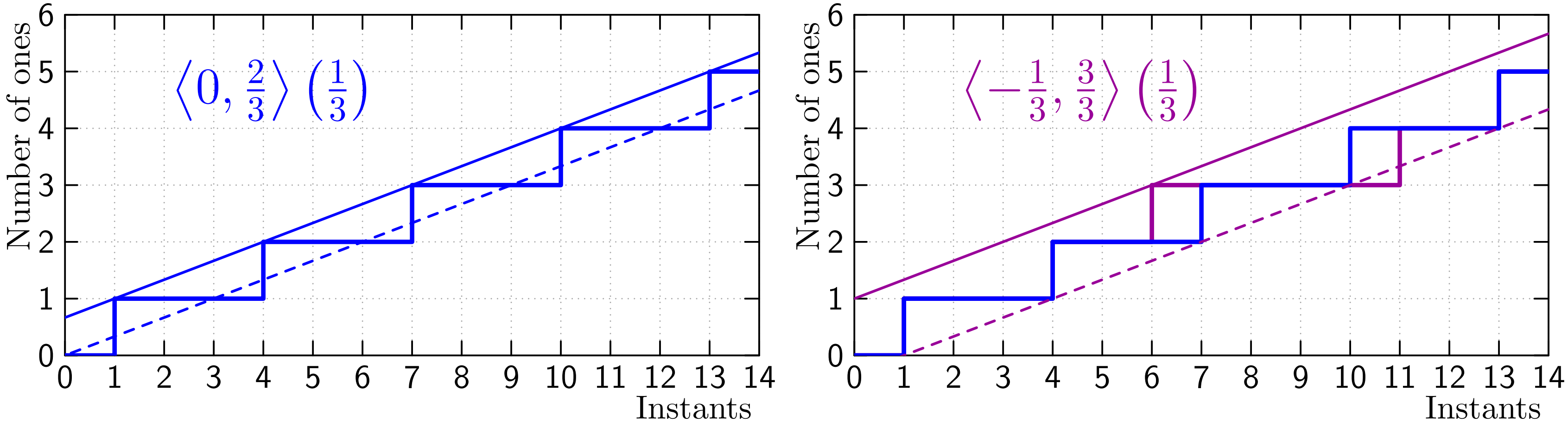
$$a_1 \ \mathit{on}^{\sim} \ a_2 = \left\langle \frac{1}{5}, \frac{7}{5} \right\rangle \left(\frac{3}{5} \right) \ \mathit{on}^{\sim} \ \left\langle -\frac{6}{5}, -\frac{2}{5} \right\rangle \left(\frac{3}{5} \right)$$

$\mathit{on}^{\sim}$ operator definition :

$$\begin{array}{rl} & \langle \ {b^0}_1 \ , \ {b^1}_1 \ \rangle (\ r_1 \ ) \\ \mathit{on}^{\sim} & \langle \ {b^0}_2 \ , \ {b^1}_2 \ \rangle (\ r_2 \ ) \\ = & \langle \ {b^0}_1 \times r_2 + {b^0}_2 \ , \ {b^1}_1 \times r_2 + {b^1}_2 \ \rangle (\ r_1 \times r_2 \ ) \end{array}$$

with $\quad {b^0}_1 \leq 0, \quad {b^0}_2 \leq 0$

Modeling Jitter



- set of clock of rate $r = \frac{1}{3}$ and jitter 1 can be specified by $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right)$
- $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right) = \langle -1, 1 \rangle (1)\ \textbf{\textit{on}}^{\sim}\ \langle 0, \frac{2}{3} \rangle \left(\frac{1}{3}\right)$
- $f :: \forall \alpha . \alpha \rightarrow \alpha\ \texttt{on}^{\sim}\ \langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right)$

Formalization in a Proof Assistant

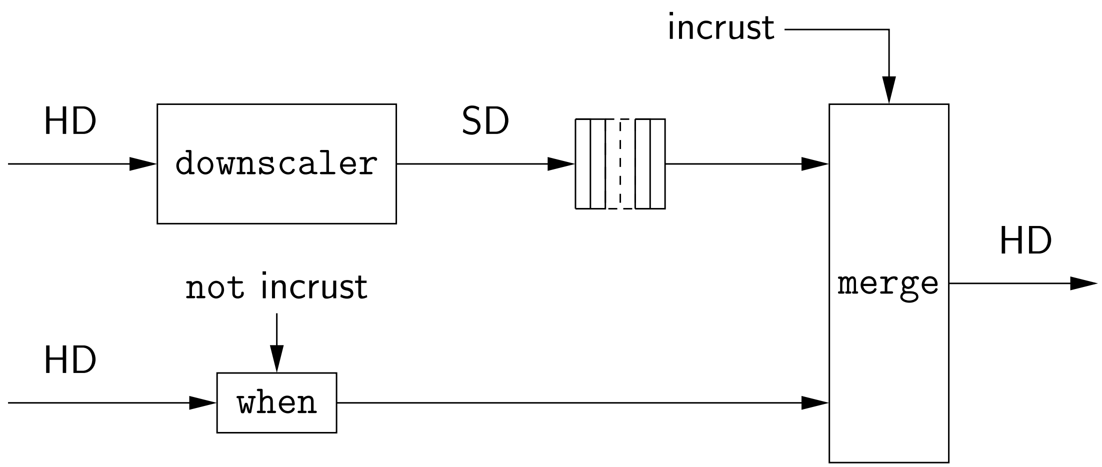
Most of the properties have been proved in Coq

- example of property

```
Property on_absh_correctness:
  forall (w1:ibw) (w2:ibw),
  forall (a1:abstractionh) (a2:abstractionh),
  forall H_wf_a1: well_formed_abstractionh a1,
  forall H_wf_a2: well_formed_abstractionh a2,
  forall H_a1_eq_absh_w1: in_abstractionh w1 a1,
  forall H_a2_eq_absh_w2: in_abstractionh w2 a2,
  in_abstractionh (on w1 w2) (on_absh a1 a2).
```

- number of Source Lines of Code
  - specifications : about 1600 SLOC
  - proofs : about 5000 SLOC

Back to the Picture in Picture Example



- abstraction of downscaler output :

$abs((10100100)\ on\ 0^{3600}(1)\ on\ (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720}))$

$= \langle 0, \frac{7}{8} \rangle \left(\frac{3}{8}\right)\ on^{\sim}\ \langle -3600, -3600 \rangle\ (1)\ on^{\sim}\ \langle -400, 480 \rangle \left(\frac{4}{9}\right) = \left\langle -2000, -\frac{20153}{18} \right\rangle \left(\frac{1}{6}\right)$

- minimal delay and buffer :

	delay	buffer size
exact result	9 598 (≈ time to receive 5 HD lines)	192 240 (≈ 267 SD lines)
abstract result	11 995 (≈ time to receive 6 HD lines)	193 079 (≈ 268 SD lines)

Conclusion

Ensuring synchronous and other static properties

- specify/check logical time as special types
- initially a dependent type system ; now an ML type system with extension by "Laufer & Odersky"
- this is the way it is done in the `Lucid Synchrone` compiler the one of SCADE 6
- some other properties can be expressed as dedicated type-systems (correct initialization of registers, causality analysis)

DSL embedding

- achieving the same result by designing a DSL (e.g., in Haskell) is difficult
- how to ensure synchrony, the absence of causality loops, unbounded FIFOs (unless we forbid non-length preserving functions) ?
- compilation through maximal static expansion does not work well when targeting software code